Security and Communication in Mobile Object Systems

Jan Vitek, Manuel Serrano and Dimitri Thanos

1 Introduction

The rapid growth of computer networks has created an opportunity for developing massively distributed computer systems. Such systems will likely consist of loose communities of heterogeneous machines running different operating systems with different security policies. The challenge is to design a reliable, and yet efficient, infrastructure trustworthy enough for electronic commerce and flexible enough to allow software upgrades as well as new functionality to propagate in a decentralized, inherently insecure, wide area network. *Mobile object systems* embody a paradigm where computations, *i.e.* running programs, may move across the network and carry out truly distributed computations. The vision is that computations structured as autonomous systems of objects will roam the network performing complex tasks on the behalf of their human owner. These mobile systems of objects carry their data as well as their code with them during their journey; thus allowing almost unlimited extendability. Such unfettered mobility raises justified security concerns. From the host's stand point first. Can an arbitrary code fragment be entrusted with local resources? To what degree is it possible to control the behaviour of downloaded code? How can secrecy and integrity be preserved? From the sender's stand point next. Is it possible to entrust the network with mobile computations that encode valuable knowledge and are empowered to carry out commercial transactions? Even though it is technically feasible to charge foreign computations for small service such as execution time or storage [39]. The key question whether there is a way to achieve a sufficient level of security for this approach to be viable? Currently, we must answer by the negative. None of the existing mobile computation systems meet the security requirements of electronic commerce. Lack of security fosters a *just say no* attitude towards mobile computations in portions of the scientific and business community [38]. The proverbial ball is now in the camp of mobile computations research. It is up to us to demonstrate that mobile objects may meet the stringent security criteria of real world applications.

The main contribution of this chapter is the study of security threats inherent to communication in mobile object systems. We will study how mobile object systems communicate. Describe the dangers of traditional communication mechanisms and outline two research directions currently being investigated. The structure of the chapter is the following. Section 2 describes security issues in mobile object systems. From this general overview we will focus on communication between object systems. Section 3 more precisely describes the threats that an object systems may be faced with. Section 4 is the heart of the chapter, it describes the shortcomings of existing communication mechanisms. In particular, we give examples that show the inadequacy of the security models of languages such as Java and Telescript (examples in Appendix). Finally, Section 5 sketches on going research.

2 Security and Mobile Object Systems

The term security has been subjected to much abuse lately. A secure programming system is not one that give the means to write secure application as argued by the likes of [17] and [40]. Instead a secure system is a programming system that prohibits insecure programs. While it is unrealistic to expect to present a comprehensive solution to the multiple security issues of real world applications, it is possible to study and solve subproblems. This is what we set out to do in this chapter. We provide examples that attest to the weaknesses of commercial environments such as Java and Telescript. We wish to emphasize before hand that even though some of these examples (but not all) may be avoided by changing coding style, the point is that it is possible, and even more, that it is easy to breach security in those environments.

A mobile object system architecture is composed of four components: (a) the host—a computer and operating system, (b) the computational environment (CE)—the run-time system, (c) mobile object systems—the computations currently running on the CE, and (d) a network or communication subsystem that interconnects CEs located on different hosts.

To establish a parallel with Java[1] the architecture maps as follows: (a) is the native operating system, for instance UNIX, (b) is a web browser, perhaps Netscape, (c) applets, (d) the Internet. In the long run the distinction between OS and CE may well disappear.

In this architecture, the computational environment is the key to overall system security. The protection mechanisms incorporated in the implementation of the CE enforce the chosen security policy. These protection mechanisms must be flexible enough to accommodate the policies of different organizations as well as comprehensive enough to ensure that the policy can not be breached [10][20] [16][21].

Protection mechanisms may be placed at all component boundaries with the goal to control and regulate interaction between components. Figure 1 lists the five boundaries that require security: (1) communication between hosts and CEs (*i.e.* computation mobility) must be secured, this to prevent disclosure of the information contained in the data portion of computations and corruption of either data or code, (2) incoming computations must be authenticated and granted access rights, this to determine on which principal's behalf the computations will execute, who should be charged and what they are allowed to do, (3) access to the host's local resources must be controlled, this to protect the host, (4) the CE must be protected from potentially malicious computations (and vice versa), this to prevent computation from by-passing the security built-in the envi-

1. Java does not support mobile computations. The language was designed for code-on-demand (remote dynamic linking) applications. Yet, it shares many concerns with mobile objects environments and may be extended to support full-fledged mobile computations [32].

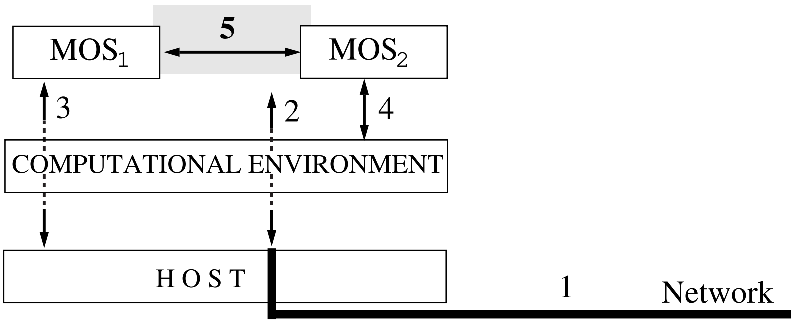

Figure 1 Security issues: (1) transfer security, (2) authentication and authorization, (3) host system security, (4) computational environment security, (5) mobile object system security.

ronment, (5) mobile object systems must be protected from each other, this to prevent a computation from disrupting another or gaining privileged information.

Transfer, Authentication and Authorization

Issues (1) and (2) are typical problems for distributed systems. It is natural to turn towards solutions developed in that field. Secure network communication and authentication require cryptographic techniques. Possible solutions include the Secure Socket Layer [14] for secure network communication, and Kerberos or some of its derivatives [31] for authentication.

Host protection

Issue (3) relates to access control mechanisms similar in purpose to those of operating systems. Yet, as Lepreau et al. [23] noted today's operating systems can not solve the problems raised by mobile code. They only provide flat protection domains while one of the implicit needs of mobile application are recursive protection domains. Recursion is needed to model the encapsulation of mobile computation within the CE and their restricted rights which must be limited to subset of the rights of the CE, *i.e*. the protection domain of the CE encapsulates the protection domains of the computations. An aspect, unrelated to OSs, which is a weak point in current environments is the *flexibility* of access control. Most Java implementation adopt an all-or-nothing approach that is not satisfactory [16][21] or even secure [9][10][22]. Flexible access control is the subject of on-going research [16][19][21] [48]. As an aside, we emphasize once more the relationship between OSs and CEs: in most cases flexible access control requires the implementation of a superset of the OS access control mechanisms, outside of the OS. For instance, Goldberg *et al.* [16] propose to curtail all system calls using the `ptrace` functionality of Solaris.

Computational environment protection

Issue (4) may be approached from different angles. The bottom line is always to prevent computations to damage their CE. Operating systems face a similar problem as they have to protect themselves from the processes they run. Unlike OSs, mobile computation architectures usually are built as single address space environments. The associated reduction in space and improved CE-computation communication speeds improves overall performance. The down side is that the CEs can not rely on address space protection mechanisms[1]. The security policy of all existing computational environments is nevertheless to restrict computations to use a well-specified interface to interact with the CE. A policy that rules out, or at least restricts, unchecked memory accesses. These are common in unsafe language such as C. To enforce the policy, the code portion of computation must be safe, code which may be either in source form, bytecode, or native object form. Code can be guaranteed safe if it is signed by a trusted party (as in the MMM Caml web browser [34]), if it carries its own proof of correctness (*e.g.* PCC [30]), if the language is safe and the bytecode can be verified to adhere to the high level language semantics (*e.g.* Java), if the code executes in a highly restricted environment and is not allowed to emit illegal instructions (*e.g.* Agent Tcl [18]), or if the code is rewritten into a safe version (*e.g.* Omniware [47]). Malice need not be confined to mobile computations.There may be even more reasons for the CE to try to subvert its computations than the converse. For instance, CEs may extract payment for service not rendered or steal the secrets of visiting computations. This raises the question whether anything can be done to provide execution and secrecy guarantees? Very little research has addressed the issue. Published results on detection of malicious CEs are so restricted that they have little practical use[2].

Mobile object system protection

Issue (5)—security at the interface between mobile computations—implies control over communication channels between computations. All communication must be regulated by protection mechanisms. The goal is that the administrator of a computational environment should be able to choose a security policy that fits the organization's security requirements and be able to impose that policy on all computations running in the CE.

This chapter will focus on security of inter-computation communication. The chapter is organized as follows. Section 2 present security requirements for communication between mobile computations. Section 4 reviews existing communication mechanisms and points out their failings. Section 5 presents two new mechanisms that remedy the problems identified in the previous section. Section 6 present our conclusions. The appendix list some example attacks.

---

1. Portions of an address space may be protected via software fault isolation (SFI) [47]. SFI introduces an overhead as memory accesses need to be checked [47]. This overhead could be reduced if the code generator was trusted as is the case of run-time code generation and just-in-time compilers.

2. [28] detects malicious platforms by duplicating computations and executing them on different hosts. After each stage results are tallied and a voting algorithm is used to pick the correct result to use in the next stage. Unfortunately, computations must be deterministic, a requirement not often met in practice.

3 Protecting Mobile Object Systems

This work departs from the main research track in security of mobile programs by shifting the attention from host security to security of mobile computations. In effect, our emphasis is on trying to control interactions between mobile objects systems. Our thesis is that there can be no overall security if computations are not secured. Such security places requirements on the design of the language and environment. This section details threats related to communication channels.

Mobile object systems

Before discussing the threat model we present the relevant characteristics of mobile object systems. An *object* is a record with fields containing references to other objects, called instance variables, and fields containing operations, called instance methods. An object has an *interface* which describes which of its fields may be accessed by other objects. Every object is an instance of a *class*. We model classes by objects. Each object has a special instance variable that refers to its class object. The variable of the class object are thus shared variables for all of the instances. We shall assume that the language is run-time safe[1]. An *object graph* is the instantaneous representation of the state of a group of objects. Very briefly, an object graph consists of a set of vertices and a set of directed edges. There are two kinds of vertices: objects, classes. We say that an object `o′` is *reachable* from `o` if there is a path from `o` to `o′` in the object graph `g`. An *object system* `obj` is the transitive closure of the object graph rooted at `obj`. This transitive closure represents the state of the system. A *computation* is a sequence of method invocations and variable accesses performed on the objects of a system `obj`. An object system is run in a *computational environment* which associates an authorization to every object system and controls the system's execution and its consumption of resources. *Mobile object systems* are object systems that may move between computational environments.

Protection domains

Any meaningful discussion of security requires a notion of protection domain. Security is relative to the entities that are to be protected. Protection domains group entities that must be protected together and are the level of granularity for protection mechanisms. In operating systems, for instance, the basic protection domain is the process. The default security policy is confinement, the OS draws a protective boundary around the set of memory pages accessible to each process. The protection mechanism is implemented in the address translation scheme that maps virtual addresses into physical ones. Most mobile environments present a single system image[2] and it is more difficult to define

1. Run time safety ensures that arbitrary memory accesses are forbidden (*e.g.* no pointer arithmetic), type casts are checked, access to array and other variable length data structures is checked for overflow. These guarantees may be obtained by a combination of run-time checks, language restriction and type checking.

2. Single system image does not necessarily imply single address space. Distributed systems such as Emerald [3] and Obliq [2] provide a global address space. Java, Telescript [40] and Agent Tcl [18] have a single address space.

protection domains in terms of memory page. In object systems, it is natural to describe protection domains in terms of object graphs. Domains may be populated implicitly or explicitly. Implicit protection domains are defined in terms of reachability in object graphs, Java leans in this direction. Explicit protection domains are defined by enumeration, Telescript falls in that category with its explicit 'owner' and 'sponsor' fields [40].

Threat model

We consider four kinds of threats in this chapter:

attack	**description**
Breach of secrecy	direct access to the state of another computation either due to a failure to enforce proper security or an insecure communication channel.
Breach of integrity	modification of the state of a computation by another computation by sending state-modifying messages to objects of the victim.
Masquerading	usurping the authority/identity of another computation for a series of actions, e.g. tricking the victim into executing some code fragment.
Denial of service	excessive consumption of a finite shared resource such as processing time, memory, or the communication subsystem.

Assuming that communication between protection domains is restricted to a set of CE-provided communication channels, preventing breaches of secrecy or integrity requires that channels be secure and never open up the objects of a protection domain to inspection or modification from another domain. In the following, we show that the strong typing and encapsulation of object-oriented languages fail to protect against such attacks as they do not preserve disjointedness of object graphs. Similarly for masquerading attacks, they can be set up by using polymorphism to inject code in other computations. Finally, denial of service attacks can be mounted by abusing the communication system[1].

Furthermore, *covert communication channels* must be prohibited. This is necessary if the security policy forbids information leaks between computations running at different levels of trust. For instance consider a scenario in which a computation requests the right to read a private file (but not to communicate over the net) and another the right to communicate over the net (but not to read private files). Both requests are acceptable from the stand point of security, as the program that has sensitive data is not able to communicate with remote machines and the program that can communicate with the outside world has no private data. The security problem is that if a covert channel can be established, the information gained by the first program may leak outside of the environment. We will give examples how to set up high-bandwidth covert channels in

1. Denial of service attacks can be mounted by using inordinate amounts of any finite resource [52]. For denial of service all that we wish to say is that use of the communication system must be accounted for. This is a failing of Telescript where it is possible to misuse the communication system [40].

Java (see also [51]). Note that discovery of low bandwidth storage and especially timing channels is an active research topic [41].

Discussion

It has been argued that object-oriented principles can be used for security [1][40][49][17]. The claim hinges on the use of encapsulation and strong typing to restrict the way a client (the program that uses an object) can interact with the object. Another claim that has been often repeated is that objects can be used as capabilities [17]. Capabilities are kinds of permits for manipulating entities, they are mostly used in operating systems [4][6][37]. It is also common to see arguments to the effect that information hiding is a form of security mechanism [17]. Morrison *et al.* argued that although objects may be used to *implement* capabilities, they are not in themselves equivalent to capabilities [29]. On a more general note the object-oriented paradigm was conceived to foster good software engineering principles. Trying to contort its features into security mechanisms is bound to fail. Our claim is that object-oriented programs are *not* more secure than programs written in any other paradigm. Some language features, such as strong typing, may help. But others, such as pervasive reference semantics, polymorphism and subtyping, are hindrances. Finally, we will show that method invocation is not a secure communication mechanism for cross domain calls. The next section reviews existing communication mechanisms and discusses their advantages and disadvantages for inter-computation communication. The last section introduces two proposals for improving communication in mobile object systems.

4 Security and Communication

Inter-computation communication mechanisms may be classified into four categories: shared memory, generative communication, datagrams, and procedure calls or in object-oriented programs method invocations. The important characteristic of communication is that it encapsulates a crossing of protection domain boundaries. From the view point of security, values which traverse such boundaries must be controlled and sharing of values between protection domains must either be forbidden or, at least, regulated.

4.1 Shared memory

Sharing memory between object systems can be done at the level of physical pages in memory, or at a slightly higher level by shared variables. Sharing of memory pages requires operating system support while the sharing of variables is implemented by the CE.

Physical sharing is a low level communication mechanism, in object systems the natural granularity is that of objects not pages. Page sharing requires that all memory accesses be checked; to be efficient this must be implemented in the operating system. This is at odds with the basic portability requirement of mobile object systems. Furthermore the interaction of shared memory with allocation strategies and garbage collection is not well understood.

Shared variables represent a more disciplined way to share memory as they are type safe and can be implemented straightforwardly in a CE that provides a global address space for computations. Sharing implies that the objects graphs of computations are not disjoint. As a communication mechanism, shared variables allow data to be exchanged between computations at no cost. This advantage is mitigated as concurrent computations usually have to synchronize their reading and writing of shared data. For security, shared variables present several problems. It is necessary to decide which computations are allowed to share variables. Often sharing decisions are static and remain in effect for the entire program execution, whereas security needs to be more dynamic as permission may be granted and revoked depending on external factors. But the main problem associated with sharing is that it does not mix well with reference semantics of objects programs. Sharing a value that has references to other values in a protection domain means that a large part of the object graph may be compromised. In fact, the object graphs of different computations may become so thoroughly intertwined that it will not be possible to ascertain if a given method invocation has a target that is within the current protection domain or if the invocation is a cross domain call. Shared variables are thus more akin to a covert channel than to a disciplined communication mechanism. This is not all. Shared variables can be used to mount secrecy, integrity, masquerading and denial of service attacks as discussed below.

Shared variables are available in most Java implementations as the computational environment (the `ClassLoader`, to be exact) loads classes only once. Thus if two applets use the same class, they will refer to the same class object and thus share all of the variables of the class (static variables in Java terminology). Applets with common classes have intersecting object graphs. The examples 1 and 2 in appendix demonstrate how easily a class variable can be hijacked and turned into a security hole. Example 3 demonstrates the difficulty of discovering covert channels. The work on information flow analysis tries to address similar problems; this work is still far from complete and its integration in a language like Java would require sever restrictions of the language [11][41][43][44][45][46]. Examples 4 and 5 are attacks that can be mounted once a protection domain has become accessible. Example 6 shows how to use shared variable to kill all running user threads in a Java CE. Example 7 shows the Telescript approach to protection based on explicit ownership checks.

4.2 Generative communication

Generative communication is a model of communication introduced by Gelernter with LINDA [15]. The generative communication model was designed to coordinate cooperating parallel computations. Computations communicate by generating new data objects, called *tuples*, and writing them in a shared data structure, called the *tuple space*, which plays the role of an associative memory. This model has been adapted for use in object-oriented programs [27][8] without addressing security issues. Not only do all problems of shared variables apply to generative communication, but there are issues related to accounting. Existing designs do not allow any form of resource accounting over tuple space resource usage.

4.3 Datagrams

Datagrams are self-contained data packets. Communication by datagrams[1] involves the exchange of unformatted packets of raw data, *e.g.* through a socket interface. Thus, to send objects or other complex data structures it is necessary to serialize the data into a portable representation and then unserialize it at the receiving end. Such serialization guarantees that the disjointedness of object graphs is preserved by datagram communication. Recall that we assumed that the language is run-time safe. One of the key requirements for run-time safety is that pointers can not be forged. This has a desirable side effect: unserialized data can not contain pointers (only references between objects serialized together are allowed). Thus there is no way to establish bridges between object systems in the same address space via datagram communication. The only security risk is that of masquerading attacks shown in example 4. This kind of attack is made possible by subtyping which allows the caller to provide subtypes of requests objects. Thus datagrams are not fully secure, a remark that applies to Java remote method invocation [50]. But the main problem of datagram communication is efficiency. For simple built-in data types the cost of datagram is at least that of copying the data twice (once into a communication buffer, and once from the buffer), plus a system call. This is already much slower than a procedure call. The cost is even higher for objects which can contain recursive structures. Objects need to be flattened at the cost of potentially multiple method invocations per object, in addition the serialization process must take care of cyclic references. This cost is not acceptable for high frequency communication.

4.4 Method invocation

Direct method invocation is the normal method of communication between objects in the same protection domain, that is within the same object system. It does seem "natural" to extend it to cross-domain communication. The argument in favour of method invocation is that with strong typing and encapsulation it is possible to restrict what a client may be able to do with an object. The weaknesses of method invocation are tied to reference semantics and subtyping. Example 5 shows that an attacker may gain access to a large portion of the victim's object graph without breaking the interface. The problem is that the interface of an object says nothing about sharing between objects. Thus strong typing is not a sufficient protection.

Once object graphs cease to be disjoint, security is basically a lost battle. For instance, an object *o* may belong to a protection domain α but executes in response from an invocation coming from an object in protection domain β. Who's authority is to be invoked? Where should memory and time consumption be charged? Telescript tried to address these issues by advocating that each object must defend itself. Thus in Telescript objects have to check the origin of messages before answering. If a message originates from a 'friend' it should be answered otherwise it should be ignored. The predictable result is an inefficient mess as each software designer must try to code coherent and comprehensive security in the objects. Furthermore, (1) changing security or composing code originating from different organizations is near to impossible, (2) validating security requires inspection of all classes, and (3) efficiency is degraded by the massive access checks, most of which unnecessary. Telescript [40] also provides 'read only' parameter, it is not clear from available documentation if this property is recursive

1. We use the term 'datagram' to avoid 'message' which is confusing in an object-oriented context.

and applies to objects reachable from the read only parameter. If it is not recursive, it is worse than useless, and if it is recursive it implies a staggering amount of run-time checking.

There is another kind of attack that uses method invocation against which strong typing fails to protect. This attack uses subtyping to pass arguments that conform to the expected types but contain dangerous implementations as shown by example 4. These kinds of masquerading attacks are not restricted to the type of the arguments. Each object `obj` given as argument to a method invocation is the root of an object graph defined by taking the transitive closure of all objects reachable from its instance variables. To prevent a masquerading attack it is necessary to guarantee that none of the objects in the graph rooted at `obj` is dangerous. This is not easy in polymorphic languages.

We would like to stress that the semantics of method invocation in languages such as Java and Telescript are fit for local communication but no to enforce security, strong typing and encapsulation notwithstanding. The problem is that method invocation knows nothing about protection domains.

4.5 Summary

We have discussed mechanisms for inter-computation communication among mobile object systems. Shared memory is too low-level and does not map well on high level abstractions. Shared variables are too undisciplined and open up systems to all kinds of threats. Generative communication has the same security weaknesses as shared variables. Datagrams are secure but inefficient as all data has to go through a costly serialization procedure. Method invocation fails to enforce security boundaries due to reference semantics of object-oriented languages and subtyping.

As a conclusion we shall compare the weaknesses of three approaches to communication. The first is the one of Java based on method invocation and shared variables. Java fails to provide any systematic security guarantees because there is no concept of program in the language. Programs, computations, or applets, are known at the level of the run-time or the operating system but not within the language. If there is no concept of computation in the language it is not surprising that there is no concept of protection domains either. Security is therefore pushed back into libraries and the run-time. This is a recipe for disaster as inconsistent policies and programming errors are bound to keep providing ways to break security [9][10]. Telescript takes a more explicit approach, there is a concept of computation and protection domain: the agent. But, all protection is dynamic and mostly in the hands of programmers. This is even worse than Java's approach to (in)security. The security code must be spread out over all classes and all applications. It is thus virtually impossible to say anything about the security of the overall system short of formally validating the code of all applications. A daunting task. As a general principle: *security of a programming system is inversely proportional to the ease of writing an insecure program*. Telescript, unlike Java, provides the means to check security but does not force these checks to be performed. Thus, we contend that Telescript security is fundamentally low. A last example is afforded by the Agent Tcl system. Although not an object-oriented language it does offer secure inter-computation communication messages based on datagrams. The basic data format is that of strings of text. The problem of this approach is speed, the overhead of communicating through strings will prevent this approach to be used in large applications that involve significant exchange of structured values.

5 Proposing Two Secure Communication Mechanisms

We now outline two communications mechanisms that are currently being implemented in the framework of the SEAL project at the University of Geneva. The goal of these mechanisms is to provide a finer control over security while remaining efficient. In particular, security should not depend on the programmer not forgetting to put checking code or on other good programming practice. We want a certain level of security to be mandatory in the system.

5.1 Sealed Method Invocation

As shown in Section 4.4, method invocation between mobile object systems fails to ensure security. Breaches of the four categories described in Section 2 may occur.

- Breach of secrecy and breach of integrity:

  These two security failures may occur for the same reason: an object system `obj1` could get a reference `ref` to data belonging to another object system `obj2` via method invocation (see appendix, example 5). If `ref` is used by `obj1` to read a value it could be a *secrecy* violation; if `ref` is used by `obj1` to write a new value, it could be an *integrity* violation.

- Masquerading and denial of service:

  Any unknown code fragment `code` executed by an object system `obj` may lead to security failures because, when executed, `code` belongs to `obj`, forming a part of `obj`. Consequently, `code` may access (read/write) any data of `obj` or may consume any system resources of `obj`. Unknown code execution is incompatible with security enforcement but unknown code execution is a paradigm advocated by object languages by the means of method invocations and subtyping. When an object invokes a method `meth` it may ignore the implementation of `meth`.

We propose an extension to the object programming paradigm that prevents attacks of the four categories. The goal of this proposal if to give programming language extensions where *un-secure programs cannot be expressed*. In that sense our approach is more ambitious that one defining a system where *secure programs can be expressed*.

Sealed object

Our proposal is based on the introduction of a special kind of object: *sealed objects*. A sealed object is an object that may use all the traditional features of the object-oriented paradigm. A sealed object may belong to a class (with the restriction that this class must not contain variable or refer to classes that do), it may have instance variables. It may point to other objects or sealed objects. It may allocate its own objects or sealed objects. Sealed objects differ from traditional objects only because they do not implement methods. Instead, they implement *sealed-methods* relying on *sealed method invocations*. Sealed method invocations differ from traditional method invocations on the following points:

- Formal parameters and results of sealed invocations are passed by deep copy. This ensures that no breach of secrecy or integrity may occur because sealed method invocations prevent any reference sharing between two communicating sealed objects. As soon as a value is concerned by a sealed method invocation, a fresh deep copy is created and passed.
- Sealed method arguments are either monomorphic or use the sealed object hierarchy for their formal parameters and for their results. This ensures that no masquerading or denial of service may occur. Restricting sealed method arguments to be monomorphic means that dynamic dispatch is not used on those arguments and thus, the executed code is as known by the caller. The monomorphic restriction is recursive on the type structure of the argument. Allowing polymorphism for the sealed object hierarchy does not compromise security because sealed objects *are* secure. Let us suppose that each sealed object is allocated, by its enclosing sealed object, some system resources (such as disk file resources, memory resources or even cpu resources). If a sealed object `obj1` received another sealed object `obj2`, `obj1` does not need to know the code executed by `obj2` because `obj1` controls via its system resources allocation the consumption of `obj2`. Furthermore, whatever `obj2` implements, it can not violate the secrecy or the integrity of `obj1` because their communications are restricted to sealed invocations.

Sealed method invocations are fast because the extra-cost of an inter mobile object system communication is just the cost of the copies. Which is much less than the cost of serialization, for example. Moreover, some of these copies can be avoided in any of the following situations: a static analysis proves that the reference to a send object is never used in the sending sealed object, a static analysis determines that the receiving object will not attempt to modify the passed object or, lastly, if the passed value is an immutable value.

Sealed objects and sealed method invocations succeed in enforcing a strict general security policy but they cannot be used to implement several specific policies. For instance, two sealed object are not allowed to use specific, more flexible, sealed method invocations. The sealed method invocation is the same for all the sealed objects. Otherwise the efficiency of the approach would be compromised.

Capsules

The deep copies can sometimes be too large, up to the entire system, and may reveal sensitive information. There is a need for sending arbitrary subgraphs which need not be entirely consistent. For this we propose the mechanism of capsules, which is related to the Octopus model of Farkas and Dearle [12][13] to the substitutions of Mira da Silva [36] and to the work on adaptive parameter passing of Lopes [25]. A capsule captures a portion of the state of an object system with the guarantee that no reference exists between the capsule's contents and the rest of the system. Thus a capsule contains an object graph that is disconnected from the rest of the system. The role of capsules in communication is crucial, as they represent the only way to exchange partial data structures between protection domains. To create a capsule, it is necessary to identify, a portion of the object graph of the application, and to unlink it from the application so that no reference remains from inside the capsule to the outside and vice versa. A capsule is specified by a root object `o` and a list of fencepost objects *Fence*. The capsule is a subgraph containing only objects which can be reached from `o` without passing through any fencepost `f` $\in$ *Fence*. Object in the set of fence posts *Fence* will be replaced by *placeholders*. Placeholders are abstract specifications of the object they replace. They are tuples: the first field is a type specification, the remaining are optional and encode additional information required to recreate the original object. Once a capsule has been constructed it is not possible to send message to the objects it contains as those objects are partially unlinked. In that sense, the contents of the capsule are passive. To use a capsule's contents, it is necessary to open the capsule and provide replacement objects for all placeholders. A capsule can be opened only once.

Capsule can capture as much state as needed, including the state of all threads of control and all the attached code, or as little as a single object. The usefulness of capsules for communication comes from the fact that they create disjoint subgraphs from the main object system. For the sake of mobility it is crucial to control very tightly the amount of data transferred. Placeholders are thus used to limit the size of the object graph to store in the capsule, and they also define the point where to reconnect the contents of the capsule to the environment.

5.2 Sealed Object Spaces

Sealed method invocation still has one minor drawback, it is a directed communication mechanism. Sometimes undirected communication or multicast communication may be desirable. Sealed method invocation is also synchronous, that is, the receiver must answer all requests in order. We propose to add generative communication as an alternative to method invocation when communication must be undirected or asynchronous. The generative communication model of LINDA was designed to coordinate cooperating parallel processes [15]. In LINDA, processes communicate by generating new data objects, called *tuples*, and writing them in a shared data structure, called the *tuple space*. This *tuple space* is an associative memory from which a process can retrieve tuples by pattern matching. We propose a new mechanism called *Sealed Object Spaces* which enhances the LINDA model with security and accounting features and shift the emphasis from coordination to communication between potentially hostile computations.

Sealed object spaces (SOSs) are purely local structures to a computational environment, in this respect they differ from the Jada proposal outlined in [8]. Multiple SOSs can coexist within a single environment. A computation may be connected to zero, one or more SOSs. It may retrieve values from an object space by pattern matching. Pattern matching relies on trying to match *tuples* with *anti-tuples*. An anti-tuple is a tuple with some "holes". An element is either a literal value or a formal. For an anti-tuple to match a tuple, all actuals of the anti-tuple must be equal to corresponding elements of the tuple. All formals must have a type which is a supertype of the corresponding element in the tuple. The process of querying a SOS proceeds as follows: (0) create an anti-tuple, (1) try to match the anti-tuple with values stored in the SOS, (2) if a match is found, bind the formals of the anti-tuple to the actuals of the tuple, (3) otherwise block, until a matching tuple is written to the object space. All SOS operations are atomic.

SOSs extend the LINDA model in two respects: keys and capsules. Keys are used to control who can retrieve a tuple and allow computations to set up fine-grain access control policies on portions of the shared space. Capsules are used to pass non-primitive objects safely.

Keys allow object spaces to be used for private communication. The principle is simple, every tuple with a field which contains a `PublicKey` can only be matched by an anti-tuple containing the corresponding `PrivateKey`. New public/private key pairs can be created and it is possible for agents to communicate private keys. These keys are objects managed by the object space. They can be viewed as capabilities in an operating system [37]. Using keys it is possible to have secret conversation, in fact a third party is not even able to determine that values were exchanged between two computations.

As SOSs use sealed method invocation, all values passed into a SOS are guaranteed to be reference free.

Accounting is under the care of the object space, which keeps track of memory consumption and time spent retrieving tuples on the behalf of a computation. Charging computations for processing time is straightforward as tuples are "passive" while in the object space. This means that unlike other proposals, matching is kept simple, in particular we do not invoke methods on the tuples or their components [27]. The memory used by each computation is equal to the size of all of its tuples still in the object space. The ownership of a tuple changes when it is input by another computation. Issues such as expiration policies for old tuples and tuple garbage collection are currently being investigated.

6 Conclusion

This chapter has investigated security in mobile object systems and focused on communication security between mobile object systems executing on the same computational environment. The conclusions that we have come to are that security measures based on strong typing and encapsulation fail to protect effectively mobile object systems from breach of integrity and secrecy, masquerading and denial of service attacks. In systems such as Telescript and Java, the choices for communicating between object systems are either to use mechanisms which are highly inefficient but secure (datagrams) or fast but insecure (shared variables, method invocation).

As a solution this chapter outlined two proposals to add security in mobile object systems. The first is to introduce sealed objects, which are objects that enforce strong security boundaries around their subobjects. The second proposal builds a secure generative communication paradigm based on sealed objects.

References

[1] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposiumon Operating Systems Principles (SOSP-15)*, pages 267–284, Coppper Mountain, CO, 1996.

[2] K. A. Bharat and L. Cardelli. Migratory applications. In *Proceedings of ACM Symposium on User Interface Software and Technology `95*, Pittsburgh, PA, Nov. 1995.

[3] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Trans. Softw. Eng.*, 13(1):65—76, Jan. 1987.

[4] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKos nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95—112. USENIX Association, April 1992.

[5] L. Cardelli. Mobile computation. Position paper, Digital SRC, 1996.

[6] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transaction on Computer Systems*, May 1994.

[7] D. Chess, B. Grosof, and C. Harrison. Itinerant agents for mobile computing. *IEEE Personal Communications*, 2(4):34 — 49, Oct. 1995.

[8] P. Ciancarini and D. Rossi: Jada: coordination and communication for Java agents. In [42].

[9] D. Dean. The security of static typing with dynamic linking. In *Fourth ACM Conference on Computer and Communications Security*, Zurich, April 1997.

[10] D. Dean, E. W. Felten, and D. S. Wallach. Java security: From Hotjava to Netscape and beyond. In *1996 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1996. IEEE, IEEE.

[11] D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504—513, July 1977.

[12] A. Farkas and A. Dearle. Octopus: A reflective language mechanism for object manipulation. In *Proceedings of the Fourth International Workshop on Database Programming Languages*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[13] A. Farkas and A. Dearle. The Octopus model and its implementation. *Australian Computer Science Communications*, 16(1), 1994.

[14] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol (version 3.0). Technical report, Netscape Communication Corporation, Mar. 1996.

[15] D. Gelernter. Linda in context. *Commun. ACM*, 32(4), Apr. 1989.

[16] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *The Sixth USENIX Security Symposium Proceedings*, pages 1—13, San Jose, California, July 1996. The Usenix Association.

[17] T. Goldstein. The gateway security model in the Java electronic commerce framework. White paper, Sun Microsystems Laboratories / Javasoft, Decemeber 1996.

[18] R. S. Gray. Agent tcl: A flexible and secure mobile-agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop*, pages 9—23, 1996.

[19] D. Hagimont, S. Krakowiak, J. Mossière, and X. R. de Pina. A selective protection scheme for the java environment. Technical Report RT-Sirac-96-12, SIRAC, 1996.

[20] B. Hailpern and H. Ossher. Extending object to support multiple interface and access control. *IEEE Transaction on Software Engineering*, 16(11):1247—1257, November 1990.

[21] T. Jaeger, A. D. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *The Sixth USENIX Security Symposium Proceedings*, pages 131—148, San Jose, California, July 1996. The Usenix Association.

[22] M. D. LaDue. Hostile applets on the horizon. 1996.

[23] J. Lepreau, B. Ford, and M. Hibler. The persistent relevance of the local operating system to global applications. In *Proceedings of the 1996 SIGOPS European Workshop*, 1996.

[24] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *Proceedings of SIGMOD '96*, Montreal, Canada, June 1996.

[25] C. V. Lopes. Adaptive parameter passing. In *Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, volume 1049 of *Lecture Notes in Computer Science*, Konazawa, Japan, March 1996. Springer-Verlag.

[26] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented DBMS. In *OOPSLA'86 Conference Proceedings*, pages 472—482, Portland, OR, September 1986. ACM.

[27] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. In *OOPSLA'88 Proceedings*, pages 276—284, Sept. 1988.

[28] Y. Minsky, R. van Renesse, F. B. Schneider, and S. D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the 1996 SIGOPS European Workshop*, July 1996.

[29] R. Morrison, A. Brown, R. Connor, Q. I. Cutts, G. Kirby, A. Dearle, J. Rosenberg, and D. Stemple. Protection in Persistent Object Systems, In *Security and Persistence*, pages 48—66. Springer-Verlag, 1990.

[30] George C. Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language (POPL'97)*, pages 106–119, Paris, France, January 1997.

[31] B. C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Procceedings of the 13th International Conference on Distributed Systems*, Pittsburgh, PA, May 1993.

[32] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz. Network-aware mobile programs. Research report, University of Maryland, 1996.

[33] R. Riggs, A. Wolrath, J. Waldo, and K. Bharat. Pickling state in the java. In *The Second Conference on Object-Oriented Technologies and Systems (COOTS) Procedings*, pages 241—250, Toronto, Canada, June 1996. USENIX Press.

[34] F. Rouaix. A Web navigator with applets in Caml. In Fifth WWW Conference, Paris, France, May 1996.

[35] A. Rudloff, F. Matthes, and J. Schmidt. Security as an add-on quality in persistent object systems. In *Second International East/West Database Workshop*, Workshops in Computing, pages 90—108, Klagenfurt, Austria, 1995. Springer-Verlag.

[36] M. Mira da Silva: Mobility and Persistence. In [42].

[37] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse cabilities in a distributed operating system. In *Proceedings Sixth International Conference on Distributed Computer Systems*. IEEE, 1986.

[38] A.S. Tanenbaum, editor, Report of the Seventh ACM SIGOPS European Workshop, Connemara, Ireland, 9-11 Sepetember 1996.http://www.cs.vu.nl/~ast/

[39] L. Tang and S. Low. Chrg-http: A tool for micropayments on the World Wide Web. In *The Sixth USENIX Security Symposium Proceedings*, pages 123 — 129. The Usenix Association, July 1996.

[40] J. Tardo and L. Valente. Mobile Agent Security. In *Proceedings of the 41th International Conference of the IEEE Computer Society (CompCon'96),* February 1996.

[41] C.-R. Tsai V. D. Gligor and C. S. Chandersekaran. On the identification of covert storage channels in secure systems. *IEEE Transactions on Software Engineering*, 16(6):569—580, June 1990.

[42] J. Vitek, C. Tschudin, (eds): Mobile Object System: A first look at mobile object-oriented programs, Springer-Verlag, 1997.

[43] D. Volpano. Provably-secure programming languages for remote evaluation. *ACM Computing Surveys*, 28A(2):electronic, December 1996.

[44] D. Volpano and G. Smith. On the systematic design of web languages. *ACM Computing Surveys*, 28(2):315—317, June 1996.

[45] D. Volpano and G. Smith. A type-based approach to program security. In *7th Int'l Joint Conference on the Theory and Practice of Software Development*, April 1997.

[46] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 28(2):1—21, 1996.

[47] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating Systems Principles*, 1993.

[48] K. M. Walker, D. F. Stern, L. Badger, K. A. Oosendorp, M. J. Petkac, and D. L. Sherman. Confining root programs with domain and type enforcement (dte). In *The Sixth USENIX Security Symposium Proceedings*, pages 21 — 36. The Usenix Association, July 1996.

[49] J. E. White. Telescript Technology: The foundation for the electronic marketplace. General Magic White Paper, General Magic, Inc. 1994.

[50] A. Wolrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *The Second Conference on Object-Oriented Technologies and Systems (COOTS) Proceedings*, pages 219—231, Toronto, Canada, June 1996. USENIX Press.

[51] C. Yoshikawa, B. Chun, and D. Culler. Web graffiti & high bandwirth covert channels using java. January 1997.

[52] C. F. Yu and V. D. Gligor. A specification and verification method for preventing denial of service. *IEEE Transactions on Software Engineering*, 16(6):581—595, June 1990.

Appendix: Security Weakness in Object-Oriented Programs

Example 1: Shared variable communication

Most current implementations of Java do not create multiple class objects. So, when a class is used in different computations, all computations use the same class and their object graphs are not disjoint. This may be considered a security hole as it creates covert communication channels between computations and permits to launch secrecy and integrity attacks. We give an example in Java.

Victim

The victim is an applet that includes a class with a `protected` class variable `last`.

```
class Victim extends Object {      // Victim is a subclass of object with a class
     protected static Victim last; // variable last that points to the last obj
created
     Victim createFromClone() {    // a creation method that takes the last victim,
        last = last.clone();       // clones it, and increases its idNum.
        last.idNum++;
     }
```

```
    private int idNum;              // idNum is an instance variable
    ...                             // there are other methods, e.g. new.
}
```

Attacker

The attacker must be aware of class `Victim`. This is straightforward as Java does not have the means to hide class definitions. An attack is mounted by subclassing the `Victim` class and adding methods for reading and writing instance variables.

```
class Attacker extends Victim {   // Attacker extends Victim with a method that
    static Victim getLast() {     // return the last Victim created.
        return last;
    }                             // and a method to set the last Victim
created.
    static void setLast(Victim l) {
       last = l;
    }
}
```

The effect achieved by this attack is that the opponent may get at portions of the object graph of the victim, either for reading, or writing. A possible defence involves changing the declaration from `protected` to `private`. This is not always possible as the class `Victim` may originate from a library or other classes may require access to that attribute. In any case, such a solution distorts the design of the class. The code for this fix is thus:

```
class Victim extends Object {
    private static Victim last;
```

An alternative, is to forbid subclassing altogether. This may not always be appropriate.

```
final class Victim extends Object {
    protected static Victim last;
```

Example 2: Blocking synchronized methods attack

This example explores another breach of integrity arising from the use of synchronized class methods. In Java, class methods may be declared synchronized to regulate concurrent execution.

Victim

The victim defines a synchronized class method.

```
class Victim extends Object {
    protected synchronized void myMethod() {
       ...
    }
```

Attacker

The attacker need only to acquire (and not release) the synchronized method to block the victim.

```
class Attacker extends Victim {
      protected synchronized void myMethod() {
         while(true);
      }
```

When the attacker calls the synchronized method `myMethod()`, the method will block itself and all other instances of the class Victim that try to call the method. A possible defence is to make the method private or to forbid subclassing altogether. Note that in this example it is the synchronization lock that is shared between different computations.

Example 3: Shared variable covert channel

This last example with shared variables demonstrates the ease of setting up covert channels. To establish confinement it is needed to be able to prevent information from flowing in unauthorized ways between applications. The point, here, is that shared variables make it virtually impossible to determine if a method invocation is a cross domain call. Consider the following intentionally simple example:

```
myObj = WriteObject new();
Things[1] = myObj;
...
myObj.write(data);
```

Is the invocation of write a source of information leakage? On the face of it, this code seems secure. Yet, the call could be a covert channel if the array `Things` was a static variable:

<u>Victim</u>

```
class Bridge {
      static Object[] Things;
      void someMethod() {
         ...
         myObj = WriteObject new();
         Things[1] = myObj;
         ...
         myObj.write(data);
      }
}
```

<u>Attacker</u>

The attacker (or accomplice) needs only define a subclass and try to catch the assignment to the array.

```
class BridgeReader extends Bridge {
      void looping() {
         ...
         thisObj = Things[1];
         thatObj = Things[1];
         while (thisObj == thatObj)
```

```
         thisObj = Things[1];
      thisObj.read() .... // covert channel
   ...
```

The method `looping` tries to read the object assigned to the first position of the `Things` array. Its success depends on the scheduling, but the point of this example is that it is *possible* to establish covert channels so that communication needs not be restricted to the identified shared variables. Furthermore, this example shows that in some cases it may be difficult to prove that a code fragment is secure.

Example 4: Masquerading attacks

Masquerading attacks may occur when the opponent is allowed to invoke methods of the target. This kind of attack is also valid across address spaces if RMI is used [50].

<u>Victim</u>

The victim contains an innocuous looking class that merely checks whether the date passed as argument corresponds to the user's birthday.

```
class DateChecker
      private Date birthday;
      public today(Date d) {
         if (d.sameDay(birthday)) {
            ...
      }
}
```

<u>Attacker</u>

The attacker needs to obtain a reference to the `DateChecker` object, and instead of passing it a date it passes an instance of class `BadDate`. This class overrides method `sameDay` to perform some malicious action with the authority of the victim.

```
class BadDate extends Date {
      public bool sameDay(Date d) {
         while(true) {
            ... do something nasty
         }
         ...
      }
}
```

The Java defence would be to define class `Date` as final. Final classes can not be subclassed, thus can not be used for masquerading attacks. Furthermore, all instance variables of `Date` must be final as well, and so on recursively. In effect forfeiting polymorphism. Of course, if `Date` or any of the classes it depends on is defined in a library this whole line of defence breaks down.

Example 5: Breaching secrecy and integrity

Breach of secrecy/integrity. This attack uses reference semantics of object applications running in the same address space to obtain a toehold in the object graph of the victim. The danger comes from that victim uses a value that belongs to another object graph to store its objects.

Victim

The victim needs only have a method that accepts some kind of container.

```
class Getter extends Object {
     private List listOfThings;
     public getList(List l) {
        listOfThings = l;
     }
}
```

Attacker

The attacker must obtain a reference to an object of type `Getter` for this kind of attack. Then by passing it a list the attacker is able to break security. This, because it is allowed to retain a reference on the object it gave.

```
class Badie
     private List watchList = new List();
     publicDoIt(Getter g) {
        g.getList(watchList);            // The attacker passes in the watchlist
        ...                              // then waits for the victim to fill it with
        watchList.doSomething();         // values and send watchlist some
message
     }
}
```

Appropriate protection is to enforce strict disjointedness of object graphs.

Example 6: Breaching integrity

Breach of integrity. Another breach of integrity can be easily set with the `Thread-Killer` class [22] which kills user threads in the Java virtual machine. This attack works because Threads are objects which are not in protected domains and the `ThreadKiller` class is able to obtain references on them.

Attacker

The attacker must code a class that does the following operation. Note the original class discussed in [22] is slightly more elaborate and does not run the risk of killing itself.

```
class ThreadKiller {
     public static void killAllThreads() {
        ThreadGroup current, top, parent;

        top = current = Thread.currentThread().getThreadGroup();
        parent = top.getParent;
```

```
      while (parent!=nil) {
         top = parent; parent = parent.getParent();
      }
      find(top);
   }
   private static void find(ThreadGroup g) {
      if (g != null) {
         int numThread = g.activeCount();
         int numGroups = g.activeGroupCount();
         Thread[] threads = new Thread[numThreads];
         ThreadGroup[] groups = new ThreadGroup[numGroup];
         g.enumerate(threads, false);
         g.enumerate(groups,false);
         for (int i = 0; i <numThreads; i++) {
            Thread t = threads[i];
            if (t != null) t.stop();
         }
         for (int i = 0; i <numGroups; i++) {
            find(groups[i]);
         }
      }
   }
```

Appropriate protection is to forbid access to objects that belong to the object graph of another applet.

Example 7: Explicit protection domains

The Telescript protection model is more elaborate than that of Java. In short, each object and method has both an owner and a sponsor. The owner is the principal to whom the object belongs and the sponsor is the principal on whose authority the object executes. Telescript provides a way to access the owner and sponsor from outside of their environment.

The secure programming style advocated in [40] boils down to the following style (expressed in Java for simplicity). The class `Protectee` is the class that should be protected, the class `Protector` implements a security policy. All methods of `Protectee` are redefined in `Protector` to check source of the call.

```
class Protectee {
     public void method_1() { ... }
     ...
}
class Protector extends Protectee {
     public void safe_method_1() throws AccessViolation {
        Sponsor sponsor = sponsor.name.authority;
        Class class = client.class;

        if (friends.find(sponsor) || okClasses.find(class) ) {
        ...
     }
}
```

The problem with this is that in general aliasing makes it quite difficult to be sure which objects actually need to be protected. This means that if any serious degree of security is required, all non-trivial objects will have to be protected. This implies a level of inefficiency that makes a system built this way unusable and a burden on programmers that is not acceptable. Finally, security is spread all over the application and can not be easily verified without validating the entire code base.